6 – Redirection

In this lesson we are going to unleash what may be the coolest feature of the command line. It's called *I/O redirection*. The “I/O” stands for *input/output* and with this facility we can redirect the input and output of commands to and from files, as well as connect multiple commands together into powerful command *pipelines*. To show off this facility, we will introduce the following commands:

- `cat` – Concatenate files
- `sort` – Sort lines of text
- `uniq` – Report or omit repeated lines
- `grep` – Print lines matching a pattern
- `wc` – Print newline, word, and byte counts for each file
- `head` – Output the first part of a file
- `tail` – Output the last part of a file
- `tee` – Read from standard input and write to standard output and files

Standard Input, Output, and Error

Many of the programs that we have used so far produce output of some kind. This output often consists of two types:

- The program's results, that is, the data the program is designed to produce
- Status and error messages that tell us how the program is getting along

If we look at a command like `ls`, we can see that it displays its results and its error messages on the screen.

Keeping with the Unix theme of “everything is a file,” programs such as `ls` actually send their results to a special file called *standard output* (often expressed as *stdout*) and their status messages to another file called s*tandard error* (*stderr*). By default, both standard output and standard error are linked to the screen and not saved into a disk file.

In addition, many programs take input from a facility called *standard input* (*stdin*), which is, by default, attached to the keyboard.

I/O redirection allows us to change where output goes and where input comes from. Normally, output goes to the screen and input comes from the keyboard, but with I/O redirection, we can change that.

Redirecting Standard Output

I/O redirection allows us to redefine where standard output goes. To redirect standard output to another file instead of the screen, we use the `>` redirection operator followed by the name of the file. Why would we want to do this? It's often useful to store the output of a command in a file. For example, we could tell the shell to send the output of the `ls` command to the file `ls-output.txt` instead of the screen:

```
[me@linuxbox ~]$ ls -l /usr/bin > ls-output.txt
```

Here, we created a long listing of the `/usr/bin` directory and sent the results to the file `ls-output.txt`. Let's examine the redirected output of the command, shown here:

```
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1 me   me   167878 2018-02-01 15:07 ls-output.txt
```

Good — a nice, large, text file. If we look at the file with `less`, we will see that the file `ls-output.txt` does indeed contain the results from our `ls` command.

```
[me@linuxbox ~]$ less ls-output.txt
```

Now, let's repeat our redirection test, but this time with a twist. We'll change the name of the directory to one that does not exist:

```
[me@linuxbox ~]$ ls -l /bin/usr > ls-output.txt
ls: cannot access /bin/usr: No such file or directory
```

We received an error message. This makes sense since we specified the nonexistent directory `/bin/usr`, but why was the error message displayed on the screen rather than being redirected to the file `ls-output.txt`? The answer is that the `ls` program does not send its error messages to standard output. Instead, like most well-written Unix programs,

it sends its error messages to standard error. Since we only redirected standard output and not standard error, the error message was still sent to the screen. We'll see how to redirect standard error in just a minute, but first let's look at what happened to our output file:

```
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1 me   me   0 2018-02-01 15:08 ls-output.txt
```

The file now has zero length! This is because when we redirect output with the “>” redirection operator, the destination file is always rewritten from the beginning. Since our `ls` command generated no results and only an error message, the redirection operation started to rewrite the file and then stopped because of the error, resulting in its truncation. In fact, if we ever need to actually truncate a file (or create a new, empty file), we can use a trick like this:

```
[me@linuxbox ~]$ > ls-output.txt
```

Simply using the redirection operator with no command preceding it will truncate an existing file or create a new, empty file.

So, how can we append redirected output to a file instead of overwriting the file from the beginning? For that, we use the `>>` redirection operator, like so:

```
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
```

Using the `>>` operator will result in the output being appended to the file. If the file does not already exist, it is created just as though the `>` operator had been used. Let's put it to the test:

```
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1 me   me   503634 2018-02-01 15:45 ls-output.txt
```

We repeated the command three times resulting in an output file three times as large.

Redirecting Standard Error

Redirecting standard error lacks the ease of a dedicated redirection operator. To redirect standard error we must refer to its *file descriptor*. A program can produce output on any of several numbered file streams. While we have referred to the first three of these file streams as standard input, output and error, the shell references them internally as file descriptors 0, 1, and 2, respectively. The shell provides a notation for redirecting files using the file descriptor number. Since standard error is the same as file descriptor number 2, we can redirect standard error with this notation:

```
[me@linuxbox ~]$ ls -l /bin/usr 2> ls-error.txt
```

The file descriptor “2” is placed immediately before the redirection operator to perform the redirection of standard error to the file `ls-error.txt`.

Redirecting Standard Output and Standard Error to One File

There are cases in which we may want to capture all of the output of a command to a single file. To do this, we must redirect both standard output and standard error at the same time. There are two ways to do this. Shown here is the traditional way, which works with old versions of the shell:

```
[me@linuxbox ~]$ ls -l /bin/usr > ls-output.txt 2>&1
```

Using this method, we perform two redirections. First we redirect standard output to the file `ls-output.txt` and then we redirect file descriptor 2 (standard error) to file descriptor 1 (standard output) using the notation `2>&1`.

Notice that the order of the redirections is significant. The redirection of standard error must always occur *after* redirecting standard output or it doesn't work. The following example redirects standard error to the file `ls-output.txt`:

```
>ls-output.txt 2>&1
```

However, if the order is changed to the following, standard error is directed to the screen.

```
2>&1 >ls-output.txt
```

Recent versions of `bash` provide a second, more streamlined method for performing this combined redirection shown here:

```
[me@linuxbox ~]$ ls -l /bin/usr &> ls-output.txt
```

In this example, we use the single notation `&>` to redirect both standard output and standard error to the file `ls-output.txt`. We can also append the standard output and standard error streams to a single file like so:

```
[me@linuxbox ~]$ ls -l /bin/usr &>> ls-output.txt
```

Disposing of Unwanted Output

Sometimes “silence is golden,” and we don't want output from a command, we just want to throw it away. This applies particularly to error and status messages. The system provides a way to do this by redirecting output to a special file called “/dev/null”. This file is a system device often referred to as a *bit bucket*, which accepts input and does nothing with it. To suppress error messages from a command, we do this:

```
[me@linuxbox ~]$ ls -l /bin/usr 2> /dev/null
```

`/dev/null` In Unix Culture

The bit bucket is an ancient Unix concept and because of its universality, it has appeared in many parts of Unix culture. When someone says he/she is sending your comments to `/dev/null`, now you know what it means. For more examples, see the Wikipedia article on `/dev/null`.

Redirecting Standard Input

Up to now, we haven't encountered any commands that make use of standard input (actually we have, but we’ll reveal that surprise a little bit later), so we need to introduce one.

cat – Concatenate Files

The cat command reads one or more files and copies them to standard output like so:

```
cat [file...]
```

In most cases, we can think of cat as being analogous to the TYPE command in DOS. We can use it to display files without paging. For example, the following will display the contents of the file ls-output.txt:

```
[me@linuxbox ~]$ cat ls-output.txt
```

cat is often used to display short text files. Since cat can accept more than one file as an argument, it can also be used to join files together. Say we have downloaded a large file that has been split into multiple parts (multimedia files are often split this way on Usenet), and we want to join them back together. If the files were named:

```
movie.mpeg.001 movie.mpeg.002 ... movie.mpeg.099
```

we could join them back together with this command as follows:

```
cat movie.mpeg.0* > movie.mpeg
```

Since wildcards always expand in sorted order, the arguments will be arranged in the correct order.

This is all well and good, but what does this have to do with standard input? Nothing yet, but let's try something else. What happens if we enter cat with no arguments?

```
[me@linuxbox ~]$ cat
```

Nothing happens, it just sits there like it's hung. It might seem that way, but it's really doing exactly what it's supposed to do.

If cat is not given any arguments, it reads from standard input and since standard input is, by default, attached to the keyboard, it's waiting for us to type something! Try adding the following text and pressing Enter:

```
[me@linuxbox ~]$ cat
```

```
The quick brown fox jumped over the lazy dog.
```

Next, type a `Ctrl-d` (i.e., hold down the Ctrl key and press "d") to tell `cat` that it has reached *end of file* (EOF) on standard input:

```
[me@linuxbox ~]$ cat
The quick brown fox jumped over the lazy dog.
The quick brown fox jumped over the lazy dog.
```

In the absence of filename arguments, `cat` copies standard input to standard output, so we see our line of text repeated. We can use this behavior to create short text files. Let's say we wanted to create a file called `lazy_dog.txt` containing the text in our example. We would do this:

```
[me@linuxbox ~]$ cat > lazy_dog.txt
The quick brown fox jumped over the lazy dog.
```

Type the command followed by the text we want to place in the file. Remember to type `Ctrl-d` at the end. Using the command line, we have implemented the world's dumbest word processor! To see our results, we can use `cat` to copy the file to stdout again.

```
[me@linuxbox ~]$ cat lazy_dog.txt
The quick brown fox jumped over the lazy dog.
```

Now that we know how `cat` accepts standard input, in addition to filename arguments, let's try redirecting standard input.

```
[me@linuxbox ~]$ cat < lazy_dog.txt
The quick brown fox jumped over the lazy dog.
```

Using the `<` redirection operator, we change the source of standard input from the keyboard to the file `lazy_dog.txt`. We see that the result is the same as passing a single filename argument. This is not particularly useful compared to passing a filename argument, but it serves to demonstrate using a file as a source of standard input. Other commands make better use of standard input, as we will soon see.

Before we move on, check out the man page for `cat`, because it has several interesting

options.

Pipelines

The capability of commands to read data from standard input and send to standard output is utilized by a shell feature called *pipelines*. Using the pipe operator | (vertical bar), the standard output of one command can be *piped* into the standard input of another.

```
command1 | command2
```

To fully demonstrate this, we are going to need some commands. Remember how we said there was one we already knew that accepts standard input? It's `less`. We can use `less` to display, page by page, the output of any command that sends its results to standard output:

```
[me@linuxbox ~]$ ls -l /usr/bin | less
```

This is extremely handy! Using this technique, we can conveniently examine the output of any command that produces standard output.

The Difference Between > and |

At first glance, it may be hard to understand the redirection performed by the pipeline operator | versus the redirection operator >. Simply put, the redirection operator connects a command with a file, while the pipeline operator connects the output of one command with the input of a second command.

```
command1 > file1
```

```
command1 | command2
```

A lot of people will try the following when they are learning about pipelines, “just to see what happens”:

```
command1 > command2
```

Answer: sometimes something really bad.

Here is an actual example submitted by a reader who was administering a Linux-based server appliance. As the superuser, he did this:

```
# cd /usr/bin
# ls > less
```

The first command put him in the directory where most programs are stored and the second command told the shell to overwrite the file `less` with the output of the `ls` command. Since the `/usr/bin` directory already contained a file named `less` (the `less` program), the second command overwrote the `less` program file with the text from `ls`, thus destroying the `less` program on his system.

The lesson here is that the redirection operator silently creates or overwrites files, so you need to treat it with a lot of respect.

Filters

Pipelines are often used to perform complex operations on data. It is possible to put several commands together into a pipeline. Frequently, the commands used this way are referred to as *filters*. Filters take input, change it somehow, and then output it. The first one we will try is `sort`. Imagine we wanted to make a combined list of all the executable programs in `/bin` and `/usr/bin`, put them in sorted order and view the resulting list:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | less
```

Since we specified two directories (`/bin` and `/usr/bin`), the output of `ls` would have consisted of two sorted lists, one for each directory. By including `sort` in our pipeline, we changed the data to produce a single, sorted list.

`uniq` - Report or Omit Repeated Lines

The `uniq` command is often used in conjunction with `sort`. `uniq` accepts a sorted list of data from either standard input or a single filename argument (see the `uniq` man page for details) and, by default, removes any duplicates from the list. So, to make sure our list has no duplicates (that is, any programs of the same name that appear in both the `/bin` and `/usr/bin` directories), we will add `uniq` to our pipeline.

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | less
```

In this example, we use `uniq` to remove any duplicates from the output of the `sort` command. If we want to see the list of duplicates instead, we add the "-d" option to `uniq` like so:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq -d | less
```

wc – Print Line, Word, and Byte Counts

The `wc` (word count) command is used to display the number of lines, words, and bytes contained in files. Here's an example:

```
[me@linuxbox ~]$ wc ls-output.txt
 7902  64566 503634 ls-output.txt
```

In this case, it prints out three numbers: lines, words, and bytes contained in `ls-output.txt`. Like our previous commands, if executed without command line arguments, `wc` accepts standard input. The "-l" option limits its output to only report lines. Adding it to a pipeline is a handy way to count things. To see the number of items we have in our sorted list, we can do this:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | wc -l
2728
```

grep – Print Lines Matching a Pattern

`grep` is a powerful program used to find text patterns within files. It's used like this:

```
grep pattern [file...]
```

When `grep` encounters a "pattern" in the file, it prints out the lines containing it. The patterns that `grep` can match can be very complex, but for now we will concentrate on simple text matches. We'll cover the advanced patterns, called *regular expressions* in Chapter 19.

Let's say we wanted to find all the files in our list of programs that had the word `zip` embedded in the name. Such a search might give us an idea of some of the programs on our system that had something to do with file compression. We would do this:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | grep zip
bunzip2
bzip2
gunzip
gzip
unzip
zip
zipcloak
zipgrep
zipinfo
zipnote
zipsplit
```

There are a couple of handy options for `grep`:

- `-i`, which causes `grep` to ignore case when performing the search (normally searches are case sensitive)
- `-v`, which tells `grep` to print only those lines that do not match the pattern.

`head` / `tail` – Print First / Last Part of Files

Sometimes we don't want all the output from a command. We may only want the first few lines or the last few lines. The `head` command prints the first ten lines of a file, and the `tail` command prints the last ten lines. By default, both commands print ten lines of text, but this can be adjusted with the `-n` option.

```
[me@linuxbox ~]$ head -n 5 ls-output.txt
total 343496
-rwxr-xr-x 1 root root       31316 2007-12-05 08:58 [
-rwxr-xr-x 1 root root        8240 2007-12-09 13:39 411toppm
-rwxr-xr-x 1 root root      111276 2007-11-26 14:27 a2p
-rwxr-xr-x 1 root root       25368 2006-10-06 20:16 a52dec
[me@linuxbox ~]$ tail -n 5 ls-output.txt
-rwxr-xr-x 1 root root        5234 2007-06-27 10:56 znew
-rwxr-xr-x 1 root root         691 2005-09-10 04:21 zonetab2pot.py
-rw-r--r-- 1 root root         930 2007-11-01 12:23 zonetab2pot.pyc
-rw-r--r-- 1 root root         930 2007-11-01 12:23 zonetab2pot.pyo
lrwxrwxrwx 1 root root           6 2016-01-31 05:22 zsoelim -> soelim
```

These can be used in pipelines as well:

```
[me@linuxbox ~]$ ls /usr/bin | tail -n 5
znew
zonetab2pot.py
zonetab2pot.pyc
zonetab2pot.pyo
zsoelim
```

`tail` has an option which allows us to view files in real time. This is useful for watching the progress of log files as they are being written. In the following example, we will look at the `messages` file in `/var/log` (or the `/var/log/syslog` file if `messages` is missing). Superuser privileges are required to do this on some Linux distributions, because the `/var/log/messages` file may contain security information:

```
[me@linuxbox ~]$ tail -f /var/log/messages
Feb  8 13:40:05 twin4 dhclient: DHCPACK from 192.168.1.1
Feb  8 13:40:05 twin4 dhclient: bound to 192.168.1.4 -- renewal in
1652 seconds.
Feb  8 13:55:32 twin4 mountd[3953]: /var/NFSv4/musicbox exported to
both 192.168.1.0/24 and twin7.localdomain in
192.168.1.0/24,twin7.localdomain
Feb  8 14:07:37 twin4 dhclient: DHCPREQUEST on eth0 to 192.168.1.1
port 67
Feb  8 14:07:37 twin4 dhclient: DHCPACK from 192.168.1.1
Feb  8 14:07:37 twin4 dhclient: bound to 192.168.1.4 -- renewal in
1771 seconds.
Feb  8 14:09:56 twin4 smartd[3468]: Device: /dev/hda, SMART
Prefailure Attribute: 8 Seek_Time_Performance changed from 237 to 236
Feb  8 14:10:37 twin4 mountd[3953]: /var/NFSv4/musicbox exported to
both 192.168.1.0/24 and twin7.localdomain in
192.168.1.0/24,twin7.localdomain
Feb  8 14:25:07 twin4 sshd(pam_unix)[29234]: session opened for user
me by (uid=0)
Feb  8 14:25:36 twin4 su(pam_unix)[29279]: session opened for user
root by me(uid=500)
```

Using the “-f” option, `tail` continues to monitor the file, and when new lines are appended, they immediately appear on the display. This continues until we press `Ctrl-c`.

`tee` – Read from Stdin and Output to Stdout and Files

In keeping with our plumbing metaphor, Linux provides a command called `tee` which

creates a “tee” fitting on our pipe. The `tee` program reads standard input and copies it to both standard output (allowing the data to continue down the pipeline) and to one or more files. This is useful for capturing a pipeline's contents at an intermediate stage of processing. Here we repeat one of our earlier examples, this time including `tee` to capture the entire directory listing to the file `ls.txt` before `grep` filters the pipeline's contents:

```
[me@linuxbox ~]$ ls /usr/bin | tee ls.txt | grep zip
bunzip2
bzip2
gunzip
gzip
unzip
zip
zipcloak
zipgrep
zipinfo
zipnote
zipsplit
```

Summing Up

As always, check out the documentation of each of the commands we have covered in this chapter. We have seen only their most basic usage. They all have a number of interesting options. As we gain Linux experience, we will see that the redirection feature of the command line is extremely useful for solving specialized problems. There are many commands that make use of standard input and output, and almost all command line programs use standard error to display their informative messages.

Linux Is About Imagination

When I am asked to explain the difference between Windows and Linux, I often use a toy analogy.

Windows is like a Game Boy. You go to the store and buy one all shiny new in the box. You take it home, turn it on, and play with it. Pretty graphics, cute sounds. After a while, though, you get tired of the game that came with it, so you go back to the store and buy another one. This cycle repeats over and over. Finally, you go back to the store and say to the person behind the counter, “I want a game that does this!” only to be told that no such game exists because there is no “market

demand” for it. Then you say, “But I only need to change this one thing!” The person behind the counter says you can't change it. The games are all sealed up in their cartridges. You discover that your toy is limited to the games others have decided that you need.

Linux, on the other hand, is like the world's largest Erector Set. You open it, and it's just a huge collection of parts. There's a lot of steel struts, screws, nuts, gears, pulleys, motors, and a few suggestions on what to build. So, you start to play with it. You build one of the suggestions and then another. After a while you discover that you have your own ideas of what to make. You don't ever have to go back to the store, as you already have everything you need. The Erector Set takes on the shape of your imagination. It does what you want.

Your choice of toys is, of course, a personal thing, so which toy would you find more satisfying?

7 – Seeing the World as the Shell Sees It

In this chapter we are going to look at some of the “magic” that occurs on the command line when we press the Enter key. While we will examine several interesting and complex features of the shell, we will do it with just one new command.

- `echo` – Display a line of text

Expansion

Each time we type a command and press the Enter key, `bash` performs several substitutions upon the text before it carries out our command. We have seen a couple of cases of how a simple character sequence, for example `*`, can have a lot of meaning to the shell. The process that makes this happen is called *expansion*. With expansion, we enter something and it is expanded into something else before the shell acts upon it. To demonstrate what we mean by this, let's take a look at the `echo` command. `echo` is a shell builtin that performs a very simple task. It prints its text arguments on standard output.

```
[me@linuxbox ~]$ echo this is a test
this is a test
```

That's pretty straightforward. Any argument passed to `echo` gets displayed. Let's try another example.

```
[me@linuxbox ~]$ echo *
Desktop Documents ls-output.txt Music Pictures Public Templates
Videos
```

So what just happened? Why didn't `echo` print `*`? As we recall from our work with wildcards, the `*` character means match any characters in a filename, but what we didn't see in our original discussion was how the shell does that. The simple answer is that the shell expands the `*` into something else (in this instance, the names of the files in the current working directory) before the `echo` command is executed. When the Enter key is

pressed, the shell automatically expands any qualifying characters on the command line before the command is carried out, so the `echo` command never saw the `*`, only its expanded result. Knowing this, we can see that `echo` behaved as expected.

Pathname Expansion

The mechanism by which wildcards work is called *pathname expansion*. If we try some of the techniques that we employed in earlier chapters, we will see that they are really expansions. Given a home directory that looks like this:

```
[me@linuxbox ~]$ ls
Desktop    ls-output.txt  Pictures  Templates
Documents  Music          Public    Videos
```

we could carry out the following expansions:

```
[me@linuxbox ~]$ echo D*
Desktop Documents
```

and this:

```
[me@linuxbox ~]$ echo *s
Documents Pictures Templates Videos
```

or even this:

```
[me@linuxbox ~]$ echo [[:upper:]]*
Desktop Documents Music Pictures Public Templates Videos
```

and looking beyond our home directory, we could do this:

```
[me@linuxbox ~]$ echo /usr/*/share
/usr/kerberos/share /usr/local/share
```

Pathname Expansion of Hidden Files

As we know, filenames that begin with a period character are hidden. Pathname expansion also respects this behavior. An expansion such as the following does not reveal hidden files.

```
echo *
```

It might appear at first glance that we could include hidden files in an expansion by starting the pattern with a leading period, like this:

```
echo .*
```

It almost works. However, if we examine the results closely, we will see that the names `.` and `..` will also appear in the results. Because these names refer to the current working directory and its parent directory, using this pattern will likely produce an incorrect result. We can see this if we try the following command:

```
ls -d .* | less
```

To better perform pathname expansion in this situation, we have to employ a more specific pattern.

```
echo .[!.]*
```

This pattern expands into every filename that begins with only one period followed by any other characters. This will work correctly with most hidden files (though it still won't include filenames with multiple leading periods). The `ls` command with the `-A` option (“almost all”) will provide a correct listing of hidden files.

```
ls -A
```

Tilde Expansion

As we may recall from our introduction to the `cd` command, the tilde character (`~`) has a special meaning. When used at the beginning of a word, it expands into the name of the home directory of the named user or, if no user is named, the home directory of the current user.

```
[me@linuxbox ~]$ echo ~
/home/me
```

If user “foo” has an account, then it expands into this:

```
[me@linuxbox ~]$ echo ~foo
/home/foo
```

Arithmetic Expansion

The shell allows arithmetic to be performed by expansion. This allows us to use the shell prompt as a calculator.

```
[me@linuxbox ~]$ echo $((2 + 2))
4
```

Arithmetic expansion uses the following form:

`$((`*`expression`*`))`

where *expression* is an arithmetic expression consisting of values and arithmetic operators.

Arithmetic expansion supports only integers (whole numbers, no decimals) but can perform quite a number of different operations. Table 7-1 describes a few of the supported operators.

Table 7-1: Arithmetic Operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division (but remember, since expansion supports only integer arithmetic, results are integers).
%	Modulo, which simply means “remainder.”
**	Exponentiation

Spaces are not significant in arithmetic expressions and expressions may be nested. For example, to multiply 5 squared by 3, we can use this:

```
[me@linuxbox ~]$ echo $(($((5**2)) * 3))
```

```
75
```

Single parentheses may be used to group multiple subexpressions. With this technique, we can rewrite the previous example and get the same result using a single expansion instead of two.

```
[me@linuxbox ~]$ echo $(((5**2) * 3))
75
```

Here is an example using the division and remainder operators. Notice the effect of integer division.

```
[me@linuxbox ~]$ echo Five divided by two equals $((5/2))
Five divided by two equals 2
[me@linuxbox ~]$ echo with $((5%2)) left over.
with 1 left over.
```

Arithmetic expansion is covered in greater detail in Chapter 34.

Brace Expansion

Perhaps the strangest expansion is called *brace expansion*. With it, we can create multiple text strings from a pattern containing braces. Here's an example:

```
[me@linuxbox ~]$ echo Front-{A,B,C}-Back
Front-A-Back Front-B-Back Front-C-Back
```

Patterns to be brace expanded may contain a leading portion called a *preamble* and a trailing portion called a *postscript*. The brace expression itself may contain either a comma-separated list of strings or a range of integers or single characters. The pattern may not contain unquoted whitespace. Here is an example using a range of integers:

```
[me@linuxbox ~]$ echo Number_{1..5}
Number_1 Number_2 Number_3 Number_4 Number_5
```

In bash version 4.0 and newer, integers may also be *zero-padded* like so:

```
[me@linuxbox ~]$ echo {01..15}
01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
[me@linuxbox ~]$ echo {001..15}
001 002 003 004 005 006 007 008 009 010 011 012 013 014 015
```

Here is a range of letters in reverse order:

```
[me@linuxbox ~]$ echo {Z..A}
Z Y X W V U T S R Q P O N M L K J I H G F E D C B A
```

Brace expansions may be nested.

```
[me@linuxbox ~]$ echo a{A{1,2},B{3,4}}b
aA1b aA2b aB3b aB4b
```

So, what is this good for? The most common application is to make lists of files or directories to be created. For example, if we were photographers and had a large collection of images that we wanted to organize into years and months, the first thing we might do is create a series of directories named in numeric “Year-Month” format. This way, the directory names would sort in chronological order. We could type out a complete list of directories, but that's a lot of work and it's error-prone. Instead, we could do this:

```
[me@linuxbox ~]$ mkdir Photos
[me@linuxbox ~]$ cd Photos
[me@linuxbox Photos]$ mkdir {2007..2009}-{01..12}
[me@linuxbox Photos]$ ls
2007-01  2007-07  2008-01  2008-07  2009-01  2009-07
2007-02  2007-08  2008-02  2008-08  2009-02  2009-08
2007-03  2007-09  2008-03  2008-09  2009-03  2009-09
2007-04  2007-10  2008-04  2008-10  2009-04  2009-10
2007-05  2007-11  2008-05  2008-11  2009-05  2009-11
2007-06  2007-12  2008-06  2008-12  2009-06  2009-12
```

Pretty slick!

Parameter Expansion

We're going to touch only briefly on parameter expansion in this chapter, but we'll be covering it extensively later. It's a feature that is more useful in shell scripts than directly

on the command line. Many of its capabilities have to do with the system's ability to store small chunks of data and to give each chunk a name. Many such chunks, more properly called *variables*, are available for our examination. For example, the variable named USER contains our username. To invoke parameter expansion and reveal the contents of USER we would do this:

```
[me@linuxbox ~]$ echo $USER
me
```

To see a list of available variables, try this:

```
[me@linuxbox ~]$ printenv | less
```

You may have noticed that with other types of expansion, if we mistype a pattern, the expansion will not take place, and the `echo` command will simply display the mistyped pattern. With parameter expansion, if we misspell the name of a variable, the expansion will still take place but will result in an empty string:

```
[me@linuxbox ~]$ echo $SUER

[me@linuxbox ~]$
```

Command Substitution

Command substitution allows us to use the output of a command as an expansion.

```
[me@linuxbox ~]$ echo $(ls)
Desktop Documents ls-output.txt Music Pictures Public Templates
Videos
```

One of my favorites goes something like this:

```
[me@linuxbox ~]$ ls -l $(which cp)
-rwxr-xr-x 1 root root 71516 2007-12-05 08:58 /bin/cp
```

Here we passed the results of `which cp` as an argument to the `ls` command, thereby

getting the listing of the `cp` program without having to know its full pathname. We are not limited to just simple commands. Entire pipelines can be used (only partial output is shown here):

```
[me@linuxbox ~]$ file $(ls -d /usr/bin/* | grep zip)
/usr/bin/bunzip2:      symbolic link to `bzip2'
/usr/bin/bzip2:        ELF 32-bit LSB executable, Intel 80386,
version 1 (SYSV), dynamically linked (uses shared libs), for
GNU/Linux 2.6.9, stripped
/usr/bin/bzip2recover: ELF 32-bit LSB executable, Intel 80386,
version 1 (SYSV), dynamically linked (uses shared libs), for
GNU/Linux 2.6.9, stripped
/usr/bin/funzip:       ELF 32-bit LSB executable, Intel 80386,
version 1 (SYSV), dynamically linked (uses shared libs), for
GNU/Linux 2.6.9, stripped
/usr/bin/gpg-zip:      Bourne shell script text executable
/usr/bin/gunzip:       symbolic link to `../../bin/gunzip'
/usr/bin/gzip:         symbolic link to `../../bin/gzip'
/usr/bin/mzip:         symbolic link to `mtools'
```

In this example, the results of the pipeline became the argument list of the `file` command.

There is an alternate syntax for command substitution in older shell programs that is also supported in `bash`. It uses *backquotes* instead of the dollar sign and parentheses.

```
[me@linuxbox ~]$ ls -l `which cp`
-rwxr-xr-x 1 root root 71516 2007-12-05 08:58 /bin/cp
```

Quoting

Now that we've seen how many ways the shell can perform expansions, it's time to learn how we can control it. Take for example the following:

```
[me@linuxbox ~]$ echo this is a     test
this is a test
```

or this one:

```
[me@linuxbox ~]$ echo The total is $100.00
The total is 00.00
```

In the first example, *word-splitting* by the shell removed extra whitespace from the `echo` command's list of arguments. In the second example, parameter expansion substituted an empty string for the value of `$1` because it was an undefined variable. The shell provides a mechanism called *quoting* to selectively suppress unwanted expansions.

Double Quotes

The first type of quoting we will look at is *double quotes*. If we place text inside double quotes, all the special characters used by the shell lose their special meaning and are treated as ordinary characters. The exceptions are `$`, `\` (backslash), and `` ` `` (back-quote). This means that word-splitting, pathname expansion, tilde expansion, and brace expansion are suppressed, but parameter expansion, arithmetic expansion, and command substitution are still carried out. Using double quotes, we can cope with filenames containing embedded spaces. Say we were the unfortunate victim of a file called `two words.txt`. If we tried to use this on the command line, word-splitting would cause this to be treated as two separate arguments rather than the desired single argument.

```
[me@linuxbox ~]$ ls -l two words.txt
ls: cannot access two: No such file or directory
ls: cannot access words.txt: No such file or directory
```

By using double quotes, we stop the word-splitting and get the desired result; further, we can even repair the damage.

```
[me@linuxbox ~]$ ls -l "two words.txt"
-rw-rw-r-- 1 me   me   18 2016-02-20 13:03 two words.txt
[me@linuxbox ~]$ mv "two words.txt" two_words.txt
```

There! Now we don't have to keep typing those pesky double quotes.

Remember, parameter expansion, arithmetic expansion, and command substitution still take place within double quotes.

```
[me@linuxbox ~]$ echo "$USER $((2+2)) $(cal)"
me 4    February 2019
Su Mo Tu We Th Fr Sa
```

```
               1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29
```

We should take a moment to look at the effect of double quotes on command substitution. First let's look a little deeper at how word splitting works. In our earlier example, we saw how word-splitting appears to remove extra spaces in our text.

```
[me@linuxbox ~]$ echo this is a     test
this is a test
```

By default, word-splitting looks for the presence of spaces, tabs, and newlines (linefeed characters) and treats them as *delimiters* between words. This means unquoted spaces, tabs, and newlines are not considered to be part of the text. They serve only as separators. Since they separate the words into different arguments, our example command line contains a command followed by four distinct arguments. If we add double quotes:

```
[me@linuxbox ~]$ echo "this is a     test"
this is a     test
```

word-splitting is suppressed and the embedded spaces are not treated as delimiters; rather they become part of the argument. Once the double quotes are added, our command line contains a command followed by a single argument.

The fact that newlines are considered delimiters by the word-splitting mechanism causes an interesting, albeit subtle, effect on command substitution. Consider the following:

```
[me@linuxbox ~]$ echo $(cal)
February 2019 Su Mo Tu We Th Fr Sa 1 2 3 4 5 6 7 8 9 10 11 12 13 14
15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
[me@linuxbox ~]$ echo "$(cal)"
   February 2019
Su Mo Tu We Th Fr Sa
               1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
```

```
24 25 26 27 28 29
```

In the first instance, the unquoted command substitution resulted in a command line containing 38 arguments. In the second, it resulted in a command line with one argument that includes the embedded spaces and newlines.

Single Quotes

If we need to suppress *all* expansions, we use *single quotes*. Here is a comparison of unquoted, double quotes, and single quotes:

```
[me@linuxbox ~]$ echo text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER
text /home/me/ls-output.txt a b foo 4 me
[me@linuxbox ~]$ echo "text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER"
text ~/*.txt {a,b} foo 4 me
[me@linuxbox ~]$ echo 'text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER'
text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER
```

As we can see, with each succeeding level of quoting, more and more of the expansions are suppressed.

Escaping Characters

Sometimes we want to quote only a single character. To do this, we can precede a character with a backslash, which in this context is called the *escape character*. Often this is done inside double quotes to selectively prevent an expansion.

```
[me@linuxbox ~]$ echo "The balance for user $USER is: \$5.00"
The balance for user me is: $5.00
```

It is also common to use escaping to eliminate the special meaning of a character in a filename. For example, it is possible to use characters in filenames that normally have special meaning to the shell. These would include `$`, `!`, `&`, spaces, and others. To include a special character in a filename we can do this:

```
[me@linuxbox ~]$ mv bad\&filename good_filename
```

To allow a backslash character to appear, escape it by typing `\\`. Note that within single

quotes, the backslash loses its special meaning and is treated as an ordinary character.

Backslash Escape Sequences

In addition to its role as the escape character, the backslash is also used as part of a notation to represent certain special characters called *control codes*. The first 32 characters in the ASCII coding scheme are used to transmit commands to teletype-like devices. Some of these codes are familiar (tab, backspace, linefeed, and carriage return), while others are not (null, end-of-transmission, and acknowledge).

Escape Sequence	Meaning
`\a`	Bell (an alert that causes the computer to beep)
`\b`	Backspace
`\n`	Newline. On Unix-like systems, this produces a linefeed.
`\r`	Carriage return
`\t`	Tab

The table above lists some of the common backslash escape sequences. The idea behind this representation using the backslash originated in the C programming language and has been adopted by many others, including the shell.

Adding the `-e` option to `echo` will enable interpretation of escape sequences. You may also place them inside `$' '`. Here, using the `sleep` command, a simple program that just waits for the specified number of seconds and then exits, we can create a primitive countdown timer:

```
sleep 10; echo -e "Time's up\a"
```

We could also do this:

```
sleep 10; echo "Time's up" $'\a'
```

Summing Up

As we move forward with using the shell, we will find that expansions and quoting will be used with increasing frequency, so it makes sense to get a good understanding of the way they work. In fact, it could be argued that they are the most important subjects to

learn about the shell. Without a proper understanding of expansion, the shell will always be a source of mystery and confusion, with much of its potential power wasted.

Further Reading

- The `bash` man page has major sections on both expansion and quoting which cover these topics in a more formal manner.
- The *Bash Reference Manual* also contains chapters on expansion and quoting: http://www.gnu.org/software/bash/manual/bashref.html